4.3 Indeterminism and Evaluation

Dienstag, 13. Juni 2017 08:30

To evaluate LPs, we use the procedural semantics:

LP S

Query G

$(G, \varnothing) \vdash_S^+ (\square, \sigma)$

- $\varnothing$: identical subst.
- $\vdash_S^+$: performs binary SLD resolution
- σ: restricted to the variables of G is the answer subst.

The definition of $\vdash_S$ still has 2 indeterminisms:

- Indeterminism 1: Which program clause K should be used for the next resolution step?
- Indeterminism 2: Which literal A_i of the query should be used for the next res. step?

For this reason, there could be several (G_2, σ_2) with $(G_1, \sigma_1) \vdash_S (G_2, \sigma_2)$.

Ex. 4.3.1. Family example

Query: ?- ancestor(Z, aline). Slide 26

$(\{\neg ancestor(Z, aline)\}, \varnothing) \vdash_S (\{\neg motherOf(Z, aline)\}, \{V/Z, X/aline\})$

$(\{\neg ancestor(Z, aline)\}, \varnothing) \vdash_S (\{\neg mO(Z, Y), \neg anc(Y, aline)\}, \{V/Z, X/aline\})$

$(\{\neg ancestor(Z, aline)\}, \varnothing) \vdash_S (\{\neg mo(Z, Y), \neg anc(Y, aline)\}, \{V/Z, X/aline\})$

Indet. 1 (choice of the prog. clause) (*)

When continuing with (*), then we have to solve Indet. 2 (choice of the next negative literal for the resolution step).

This could lead to the answer subst. $\{Z/renate\}$ or to answer subst $\{Z/susanne\}$.

⇒ Indet. 1 influences the result

The computation could also be non-terminating (the rightmost path of the comput. tree is infinite)

⇒ Indet. 2 influences the termination behavior.

To improve efficiency, we do not want to build up the whole computation tree.

We first solve Indet. 2, i.e., we fix a strategy which decides which literal of the query has to be solved next (e.g., only perform resolution with the leftmost literal of any query).

This would make the search tree much smaller.

Does this affect the completeness of

binary SLD resolution (i.e., would this reduced search tree still contain all possible solutions)?

It turns out that this restriction is still complete, i.e., the reduced search tree still contains all solutions.
So one can take any selection strategy which selects the next literal of queries.
(Reason for the name "SLD resolution").

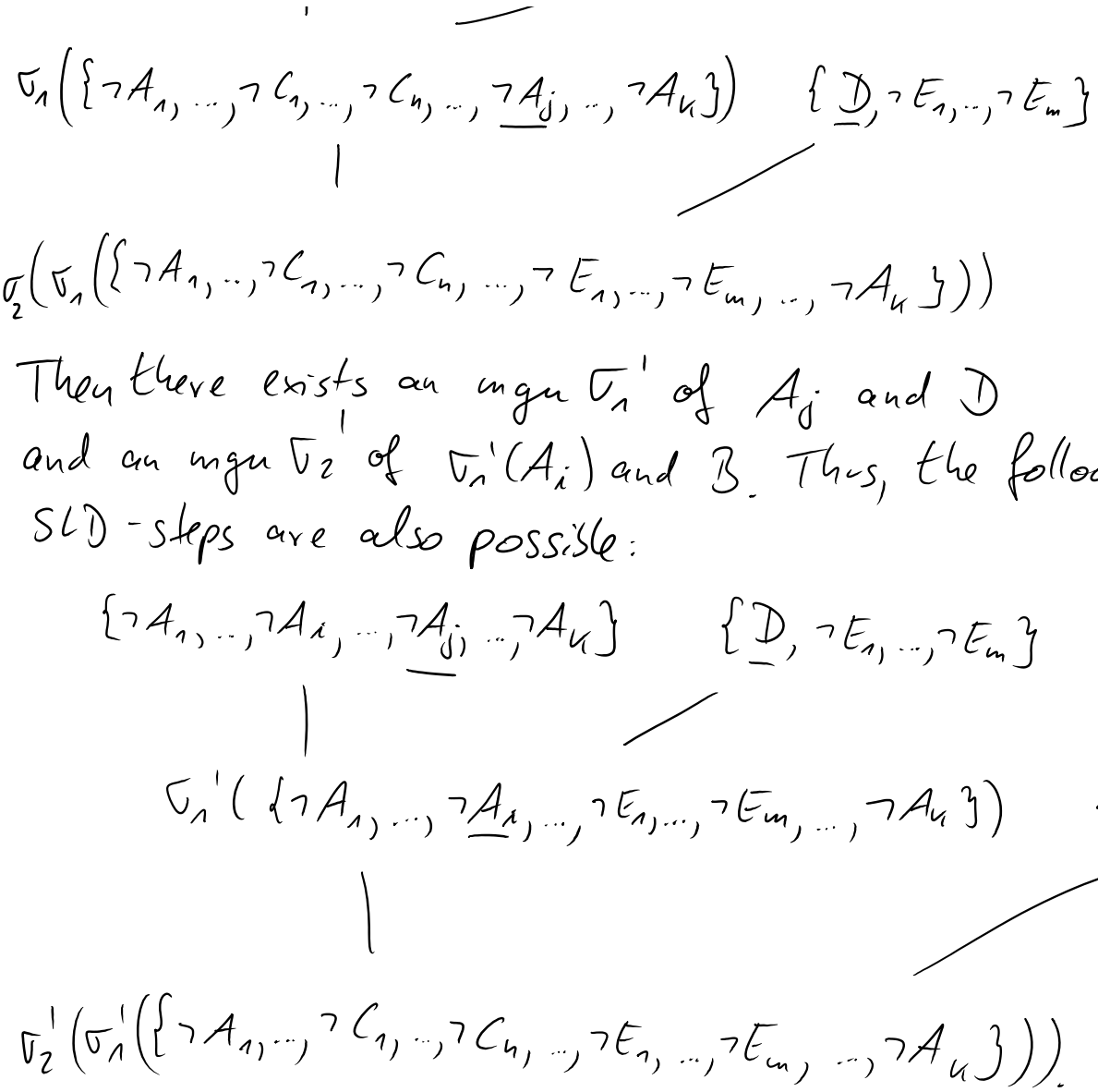
↑
selection

Reason: Exchangement Lemma
If one can first do a resolution step with $\neg A_i$ and then with $\neg A_j$, then one could also exchange these two steps and one obtains the same result.

Lemma 4.3.2. (Exchangement Lemma) Slide 25

Let $A_1, \ldots, A_k, B, C_1, \ldots, C_n, D, E_1, \ldots, E_m \in At(\Sigma, \Delta, \mathcal{V})$, where $\{\neg A_1, \ldots, \neg A_k\}, \{B, \neg C_1, \ldots, \neg C_n\}, \{D, \neg E_1, \ldots, \neg E_m\}$ are pairwise variable-disjoint. Let σ_1 be mgu of A_i and B and let σ_2 be the mgu of $\sigma_1(A_j)$ and D. So the following SLD-steps are possible:

$\{\neg A_1, \ldots, \underline{\neg A_i}, \ldots, \neg A_j, \ldots, \neg A_k\}$ $\quad$ $\{\underline{B}, \neg C_1, \ldots, \neg C_n\}$



Ex. 4.3.3 Illustrates the exchangement lemma.

```
p(Z,Z) :- r(Z).
q(W).
```

Query: `?- p(X,Y), q(X).`

$(\{\neg p(X,Y), \neg q(X)\}, \varnothing) \vdash_S (\{\neg r(Z), \neg q(Z)\}, \{X/Z, Y/Z\})$

$\vdash_S (\{\neg r(Z)\}, \{W/Z, X/Z, Y/Z\})$

Here, we first solved literal 1 and then literal 2.
By the exchangement lemma, then one could also first solve literal 2 and then literal 1, and get the same result.

$(\{\neg p(X,Y), \neg q(X)\}, \varnothing) \vdash_S (\{\neg p(W,Y)\}, \{X/W\})$

$\vdash_S (\{\neg r(Y)\}, \underbrace{\{W/Y, Z/Y\} \circ \{X/W\}}_{\{W/Y, Z/Y, X/Y\}})$

Indeed, the results are the same up to the variable renaming $\nu = \{Y/Z, Z/Y\}$.

Proof of the exchangement lemma (Lemma 4.3.2):

Since the clauses are variable-disjoint, the mgu σ_1 of A_i and B does not modify D, i.e., $\sigma_1(D) = D$.
Then for the mgu σ_2 of $\sigma_1(A_j)$ and D, we have

$$\sigma_2(\sigma_1(A_j)) = \sigma_2(D) = \sigma_2(\sigma_1(D)).$$

Thus, A_j and D are unifiable and $\sigma_2 \circ \sigma_1$ is a unifier. Therefore, there also exists an mgu σ_1' of A_j and D. Hence, there exists a substitution σ with

(*) $\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1'$ (every unifier can be obtained as an instance of the mgu).

So we can indeed start with a resolution step on $\neg A_j$ and D. Now we have to show that afterwards, a resolution step on $\sigma_1'(\neg A_i)$ and B would be possible, i.e., that $\sigma_1'(A_i)$ and B unify.
This holds, because σ is a unifier of $\sigma_1'(A_i)$ and B:

$$\begin{aligned}
\sigma(\sigma_1'(A_i)) &= \sigma_2(\sigma_1(A_i)) && \text{by } (\ast)\\
&= \sigma_2(\sigma_1(B)) && \text{as } \sigma_1 \text{ is unifier of } A_i \text{ and } B\\
&= \sigma(\sigma_1'(B)) && \text{by } (\ast)\\
&= \sigma(B) && \text{as } \sigma_1' \text{ is mgu of } A_j \text{ and } D \text{ and therefore, it does not modify } B \text{ due to variable-disjointness.}
\end{aligned}$$

As σ is a unifier of $\sigma_1'(A_i)$ and B, they also have an mgu σ_2'. Thus, there exists a subst. δ with

$$(\ast\ast) \qquad \sigma = \delta \circ \sigma_2'.$$

It remains to show that $\sigma_2 \circ \sigma_1$ and $\sigma_2' \circ \sigma_1'$ are the same up to variable renaming. To show this, we prove that $\sigma_2 \circ \sigma_1$ is an instance of $\sigma_2' \circ \sigma_1'$ and that $\sigma_2' \circ \sigma_1'$ is an instance of $\sigma_2 \circ \sigma_1$.

To show that $\sigma_2 \circ \sigma_1$ is an instance of $\sigma_2' \circ \sigma_1'$:

$$\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1' \qquad \text{by } (\ast)$$

$= \delta \circ \sigma_2' \circ \sigma_1'$ by (* *)

In a similar way, one can also show the other direction (course notes). ∎

The exchangement implies that we can use an arbitrary selection strategy for the next literal. Then every solution (path to $\square$) can still be found with the same answer subst. up to variable renaming. In Prolog, one always selects the leftmost literal.

Def 4.3.4 (Canonical Computation)
A computation $(G_1, \sigma_1) \vdash_P (G_2, \sigma_2) \vdash_P \ldots$ is canonical iff in every resolution step one performs resolution with the leftmost literal of the query G_i.

Thm 4.3.5 (Solving Indet. 2)
Let P be a LP, let G be a query. For every computation $(G, \varepsilon) \vdash_P^+ (\square, \sigma)$, there exists a canonical computation $(G, \varepsilon) \vdash_P^+ (\square, \sigma')$ of the same length, where σ and σ' are the same up to variable

renaming.

Proof Sketch: Consequence of the exchangement lemma, because one can exchange the order of the resolution steps in the original computation until it becomes canonical.
(Technical proof in course notes). ☞

This means that binary SLD resolution is still complete when restricting ourselves to canonical computations.

Ex. 4.3.6. In the computation tree of Ex. 4.3.1 we can now remove all non-canonical computations without losing any solution. Slide 27

In this example, the infinite tree even becomes finite. This tree is called SLD tree.

Indet 2 influences the term. behavior.
Since we know that Prolog only performs canonical computations, this should be taken into account by the programmer to avoid undesired non-termination.

Ex. 4.3.7

Ex. 4.3.7

```
p :- p.
q(a).
```

?- q(b), p. terminates in Prolog, because there is no canonical comp. step starting in $(\{\neg q(b), \neg p\}, \varnothing)$.

In contrast, the following query is non-terminating:

?- p, q(b).

Here, we have the canonical computation:

$(\{\neg p, \neg q(b)\}, \varnothing) \vdash_S (\{\neg p, \neg q(b)\}, \varnothing) \vdash_S \ldots$

Def 4.3.8 (SLD Tree)

Let S be a LP, let G be a query. The SLD Tree of S w.r.t. G is a finite or infinite tree whose nodes are labeled with sequences of atomic formulas and whose edges are labeled with substitutions.

The SLD tree is the smallest tree with:

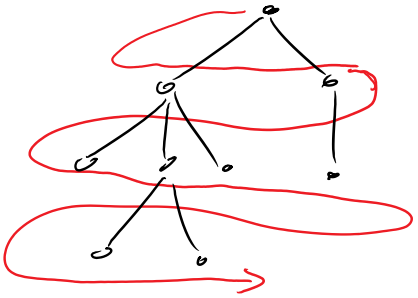
- If $G = \{\neg A_1, \ldots, \neg A_k\}$, then the root of the tree is labeled with $A_1, \ldots, A_k$.

- If a node is labeled with $B_1, \ldots, B_n$, and B_1 is unifiable with the positive literals of k variable-renamed program clauses $K_1, \ldots, K_k$ (where the clauses appear in this order in the program), then the node has k children. The i-th child is labeled with the atoms that result from a canonical computation step using resolution with K_i. So if this computation has the form $(\{\neg B_1, \ldots, \neg B_n\}, \varepsilon) \vdash_S (\{\neg C_1, \ldots, \neg C_m\}, \sigma)$, then the i-th child is labeled with $C_1, \ldots, C_m$ and the edge to this child is labeled with σ (restricted to the variables in $B_1, \ldots, B_n$).

- The answer substitutions can be obtained from the paths ending in $\square$.
If the path is labeled by $\delta_1, \delta_2, \ldots, \delta_d$ then the answer subst. is $\delta_d \circ \ldots \circ \delta_2 \circ \delta_1$ restricted to the variables of the original query G.
- Besides successful paths there can also be paths that end in a non-empty clause ("finite failure") and infinite paths.

To solve 'indeterminism', we have to determine the strategy that is used to build up the tree (and to search for □).

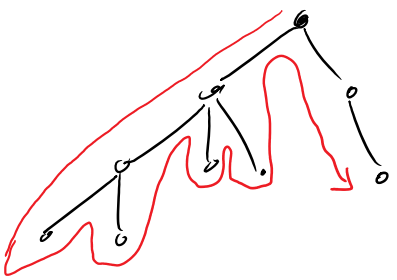
Possibility 1: Breadth-First Search



Advantage: complete technique
(it finds every □ in the SLD tree after finitely many steps ⇒ semi-decision procedure)

Disadvantage: inefficient

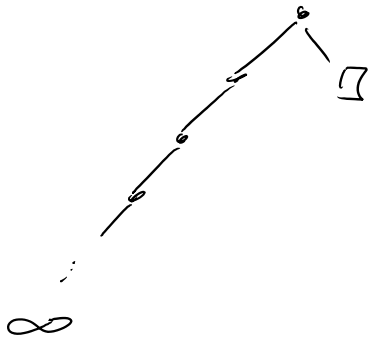
Possibility 2: Depth-First Search



Advantage: can be efficient if the solution is near the leftmost path

Disadvantage: not complete

→ programmer has to take the search strategy into account



In Prolog:

- depth-first search
- stops as soon as $\square$ is found
- If the programmer enters ";", then one continues until the next $\square$ is found.

Ex 439 To demonstrate the effect of the order literals, we exchange the 2 literals in the body of the ancestor-rule. Slide 28

Now the SLD-tree is infinite (non-termination if one continues the search after the 2nd solution).

Heuristic: perform recursive calls only if arguments are sufficiently instantiated

Ex. 4.3.10 To demonstrate the effect of the order of prog. clauses, we now exchange the Slide 29

two ancestor rules. Now Prolog's search strategy immediately goes into the infinite path and does not terminate, i.e., it does not find any solution.

Heuristic: non-recursive clauses for a predicate p should come before recursive clauses for p.

Prolog is not completely declarative, but one does have to consider its evaluation strategy.